# Refactorings of Design Defects using Relational Concept Analysis

Naouel Moha[1], Amine Mohamed Rouane Hacene[2], Petko Valtchev[3], and Yann-Gaël Guéhéneuc[1]

[1] DIRO, University of Montréal, CP 6128, Montréal, H3C 3J7, Canada
[2] LORIA, BP 239 - 54506 Vandoeuvre-lès-Nancy, Cedex France
[3] LATECE, Université du Québec à Montréal, CP 8888, Montréal, H3C 3P8, Canada

**Abstract.** Software engineers often need to identify and correct design defects, *i.e.*, recurring design problems that hinder development and maintenance by making programs harder to comprehend and/or evolve. While detection of design defects is an actively researched area, their correction — mainly a manual and time-consuming activity — is yet to be extensively investigated for automation. In this paper, we propose an automated approach for suggesting defect-correcting refactorings using relational concept analysis (RCA). The added value of RCA consists in exploiting the links between formal objects which abound in a software re-engineering context. We validated our approach on instances of the *Blob* design defect taken from four different open-source programs.

*Keywords:* Design Defects, Refactoring, Relational Concept Analysis.

## 1 Introduction

Design defects are "bad" solutions to recurring design problems that generate negative consequences on the quality characteristics of object-oriented (OO) software systems, such as evolvability and maintainability, and therefore increase the cost of software development [5,16]. Design defects, such as antipatterns [28] (e.g., the Blob addressed below), are distinguished from low-level defects, such as code smells [5] (e.g., long methods and large classes). Automatic detection and correction of design defects are thus keys for the improvement of software quality.

We proposed a systematic method to specify design defects consistently and precisely and to generate detection algorithms from their specifications automatically [17]. We specified a language based on rules that allows to define these specifications with structural, semantic, and measurable properties that characterize a design defect. This method was a first step towards the systematic detection of design defects. Yet both detection and correction of such defects are time-consuming and error-prone activities hence leaving room for automated techniques and tools. On the one hand, approaches exist for detecting design defects, for instance, using metrics [15,21], coupled with visualisation tools [13,14] and/or structural data [9]. On the other hand, to the best of our knowledge,

none of them attempts to correct discovered defects in a semi- or fully automated manner.

Thus, design defects are still dealt with manually through tedious code analyses and transformations, which divides into three main steps, possibly repeated through trials and errors: (1) Identification of the modifications to correct the design defects, (2) Application of the modifications on the program, (3) Evaluation of the resulting modified program. Step two of correction has been made easier by the recent introduction of *refactorings* [5], *i.e.*, changes performed on the source code of a program to improve its internal structure without changing its external behaviour. Thus, possible transformations are now well understood and documented and the emphasis lies on step one, *i.e.*, the decision of which modifications (or refactorings) to apply.

In the literature, Trifu *et al.* [27] proposed correction strategies mapping design defects to possible solutions. However, a solution is only an example of how the program *should have been implemented* to avoid a defect rather than a list of steps that a software engineer could follow to correct the defect. Huchard and Leblanc [11] used formal concept analysis (FCA) to suggest class hierarchy restructuring so as to maximise the sharing of specifications and code and to remove code smells (see [6] for a broader discussion on the restructuring of class hierarchies through FCA). In summary, both approaches address important issues with design defects but none attempts *to suggest refactorings to correct them.*

We propose to apply RCA, that extends FCA with the processing of individuals with links, on a suitable representation of a program to help identify appropriate refactorings for specific design defects. In particular, we examine the benefits of RCA for the correction of a very common design defect, the Blob [28, p. 73–83], also known as *God Class* [22]. The Blob reveals a procedural design (and thinking) implemented with an OO programming language. It manifests itself through a large controller class that plays a God-like role in the program by monopolizing the computation, and which is surrounded by a number of smaller data classes providing many attributes but few or no methods.

Blobs are common and RCA is particularly well-suited to suggest refactorings to correct them. Indeed, correcting a Blob amounts to splitting the Blob class into smaller cohesive sets by grouping class members that collaborate to realize a specific responsibility of the Blob class. In our context, cohesive sets are identified using formal concepts whose intents involve both proper characteristics and inter-member links, such as calls between methods. Our enhanced approach is illustrated using a running example of a library management system, which includes a Blob.

The present work builds upon a previous study described in [18] that relied on standard FCA. Its contribution is three-fold. First, a more powerful approach is adopted based on finer and richer modeling of the problem through RCA. Then, a set of enhanced rules for candidate class extraction out of the concept sets is designed, each rule is provided with an effective algorithm. Third, a mechanism to automatically interpret the results is introduced to suggest the refactorings to apply. A validation thereof involving Blobs from four different open-source

programs is also presented. The results show that RCA suggests a high rate of relevant refactorings and we briefly discuss the application of our method on further design defects.

The paper starts by a short presentation of design defects correction (Section 2). Follow a summary on RCA (Section 3) and the description of our approach (Section 4). Section 5 presents the results of a preliminary empirical study of the approach validity. Related work is summarised in Section 6 while future research directions are given in Section 7.

## 2 The Defect Correction Problem

In the following, we relate design defects to general quality criteria for OO designs using an occurrence of the Blob as running example. The defects are shown to decrease scores on these criteria. The improvement brought by the FCA-based refactorings is discussed in later sections.

### 2.1 Quality Criteria

Design defects are the results of *bad* practices that transgress *good* OO principles. Thus, we use the degree of satisfaction of those principles before and after the correction as a measure of improvement. We rely on quantification of *coupling* and *cohesion*, which are among the most widely acknowledged software quality characteristics, keys for maintainability [2].

The cohesion of a class reflects *how closely the methods are related to the instance variables in the class* [4] and is typically measured by the LCOM metric (Lack of COhesion Metric: between 0 and 1) which uses *the number of disjoint sets of methods* [4]. A low LCOM score characterises a cohesive class whereas a value close to 1 indicates a lack of cohesion and suggests the class might better be split into cohesive sets. The coupling of a class to the rest of a program is defined as the degree of its reliance on services provided by other classes [4]. It is measured by the CBO metric (Coupling Between Objects) [3] that counts the classes to which a class is coupled. A well-designed program exhibits *high* average cohesion and *low* average coupling, but it is widely known that these criteria are antinomic hence a trade-off is usually sought.

### 2.2 Further Design Defects

We choose to illustrate our approach with the Blob because it impacts negatively both cohesion and coupling: blobs show low cohesion and high coupling. Moreover, it is a frequent defect in OO programs. For example, a previous study revealed 1,146 Blobs in the Eclipse IDE [19] even though it is recognised for the quality of its design.

Yet, we found that a good number of other design defects are infected by low cohesion and high coupling, e.g., Divergent Change [5, page 79], Feature Envy [5, page 80], Inappropriate Intimacy [5, page 85], Lazy Class [5, page 83],

Shotgun Surgery [5, page 80], or Swiss Army Knife [28, page 197]. Therefore, our approach could be adapted to these defects.

### 2.3 Running Example

Our running example (see Figure 1) was inspired by a simple library management system, which includes a Blob (described in [28]). The large controller class is the class `Library_Main_Control` that accesses to data of the two surrounding data classes `Book` and `Catalog`.

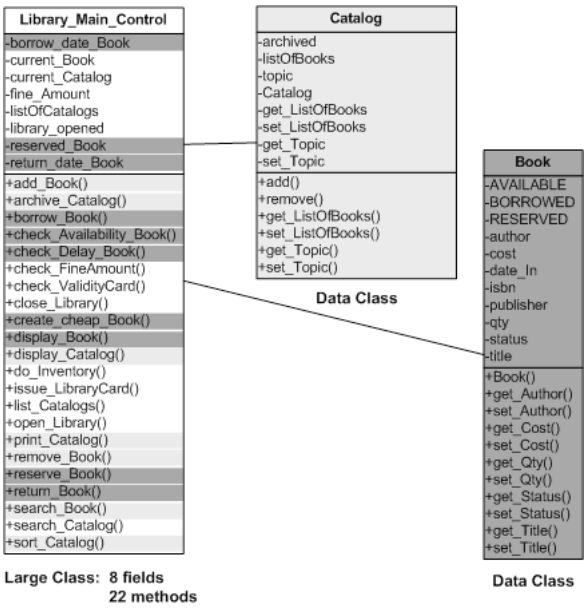


**Fig. 1.** Library Blob class diagram.

Refactoring a Blob consists in moving class members away from the large controller class to its surrounding classes or to new specifically designed classes. For the class `Library_Main_Control`, we notice that all methods and fields related to `Book` or `Catalog` could be moved to their respective data classes. As a result, data classes gain more behaviour while the large class becomes less complex. However, the process of choosing and applying refactoring is long and tedious: software engineers need to go through all methods and fields of the large class to identify the subsets thereof that form consistent cohesive sets. Yet it is a necessary pain because the result of the process may substantially improve the quality of the program.

## 3 Relational Concept Analysis

FCA offers a framework to derive conceptual hierarchies from sets of individuals based on the properties that these individuals share[4].

### 3.1 Formal Concept Analysis

FCA describes (formal) concepts both extensionally and intentionally, *i.e.*, as sets of individuals and sets of shared properties, and organizes them hierarchically—according to a generality relation—into a complete lattice, called the concept lattice. The lattice structure allows easy navigation and search as well as optimal representation of information comparable to the classical OO requirement of maximal factorisation (each property/individual is canonically represented by a unique concept). For instance, the table on the left-hand side of Fig. 2 illustrates a binary context derived from the class `Library_Main_Control` where individuals are the Blob methods while properties are methods and accessed fields[5]. Fig. 3 depicts a simplified (reduced) labeling of the concept lattice derived from this context, yet enriched by additional properties as described later in this section.

| | 'add_Book()' | 'borrow_Book()' | 'check_Availability_Book()' | 'check_FineAmount()' | 'close_Library()' | 'issue_LibraryCard()' | 'open_Library()' | 'remove_Book()' | 'reserve_Book()' | 'return_Book()' | 'search_Book()' | 'sort_Catalog()' | R-current_book | R-fine_amount | W-borrow_date_book | W-library_opened | W-reserved_book | W-return_date_book |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| add_Book() | X | | | | | | | | | | | | X | | | | | |
| borrow_Book() | | X | | | | | | | | | | | X | | X | | | X |
| check_Availability_Book() | | | X | | | | | | | | | | X | | | | | |
| check_FineAmount() | | | | X | | | | | | | | | | X | | | | |
| close_Library() | | | | | X | | | | | | | | | | | X | | |
| issue_LibraryCard() | | | | | | X | | | | | | | | | | | | |
| open_Library() | | | | | | | X | | | | | | | | | X | | |
| remove_Book() | | | | | | | | X | | | | | | | | | | |
| reserve_Book() | | | | | | | | | X | | | | | | | | X | |
| return_Book() | | | | | | | | | | X | | | | | X | | | X |
| search_Book() | | | | | | | | | | | X | | | | | | | |
| sort_Catalog() | | | | | | | | | | | | X | | | | | | |

| | add_Book() | check_Availability_Book() | check_FineAmount() | remove_Book() |
|---|---|---|---|---|
| | | | | |
| | | X | | |
| | | | | |
| | | | | |
| | | | | |
| | | | X | |
| | | | | |
| | | | | |
| | | | | |
| | | X | | |
| | | | | |
| | | | | |
| | X | | | X |

**Fig. 2. Left:** Context of methods. **Right:** Binary relation 'call' between methods.

Formal concepts naturally endow "cohesiveness" because their extents comprise members sharing *all* the properties in the respective intents. Conversely,

[4] We use *individuals* for *objects* and *properties* for *attributes* to avoid confusion with OO objects and attributes.

[5] The prefixes `R-` and `W-` that appear in the field names specify the access mode, *i.e.*, read and write, respectively.

concept extents are *maximal* sets for the respective intent. In order to identify highly cohesive sets that could jointly replace the Blob class and hence improve the overall quality, a suitable formalization consists in using class methods as individuals and instance variables as properties. For example, the concept ({`open_Library()`, `close_Library()`},{`W-library_opened`}) (concept $c9$ in Fig. 3) could generate a smaller, hence more cohesive, class.

Furthermore, in an attempt to reduce coupling in the resulting OO code, we consider the links between class members such as method calls (see Fig. 2, on the right). For instance, both methods `borrow_Book()` and `reserve_Book()` call `check_Availability_Book()`. Assigning the first two methods to the same class inevitably decreases the class coupling in the OO code. Therefore, we would like to define an approach that allows grouping these methods. We use RCA to do so because grouping individuals based on the links they share, *i.e.*, the calls of same or comparable methods, is beyond the scope of classical FCA.

### 3.2 Bringing Relations to Concept Intents

Relational concept analysis (RCA) is an approach for extracting formal concepts from sets of individuals described by properties, called also *local properties*, and links. RCA comes up with formal concepts that are connected in the same way that description logics concepts are connected, *i.e.*, by means of role restrictions involving logical quantifiers. RCA input data is organized within a structure called *relational context family* (RCF) that comprises a set of binary contexts $\mathcal{K}_i = (O_i, A_i, I_i)$ and set of binary relations $r_k \subseteq O_i \times O_j$, where $O_i$ and $O_j$ are the individual sets of $\mathcal{K}_i$ (domain) and $\mathcal{K}_j$ (range), respectively. For instance, the context encoding the access of fields by methods and the binary relation *call* that links methods of the Blob with one another form a sample RCF (see Fig. 2).

A scaling mechanism is used to translate links into context properties: relations are interpreted as features whose values are sets of individuals, hence the target properties are predicates describing these sets. The predicates are derived from the available concept lattice on the underlying context. Thus, for a given relation seen as a function $r : O_i \rightarrow 2^{O_j}$, new properties, called *relational*, of the form $qr{:}c$, are added to $\mathcal{K}_i$, where $c$ is concept on $\mathcal{K}_j$ and $q$ a scaling operator (comparable to role restriction connectors in description logics). An individual $o \in O_i$ gets a property $qr{:}c$ depending on the relationship between its link set $r(o)$ and the extent of $c = (X, Y)$. The relationship can be either inclusion, *i.e.*, $r(o) \subseteq X$ (called *universal* scaling schema, $q$ is $\forall$), or non-empty intersection, *i.e.*, $r(o) \cap X$ (called *existential* scaling schema, $q$ is $\exists$). Formally, given a context $\mathcal{K}_i{=}(O_i, A_i, I_i)$, a relation $r \subseteq O_i \times O_j$ and the lattice $\mathcal{L}_j$ of $\mathcal{K}_j$, the image of $\mathcal{K}_i$ for the existential scaling operator is: $sc_{\exists}(\mathcal{K}_i) = (O_i, A_i^+, I_i^+)$, where $A_i^+ = A_i \cup \{\exists r : c | c \in \mathcal{L}_j\}$ and $I_i^+ = I_i \cup \{(o, \exists r : c) | o \in O_i, c = (X, Y) \in \mathcal{L}_j, r(o) \cap X \neq \emptyset\}$). In the present study, as in the vast majority of software engineering applications of RCA, current or anticipated, only the existential scaling is suitable. Hence we shall be systematically omitting the $\exists$ sign in attribute names to keep notations simple.

| | *call:c0* | *call:c2* | *call:c4* | *call:c5* | *call:c6* | *call:c11* |
|---|---|---|---|---|---|---|
| `borrow_Book()` | | × | × | × | | |
| `issue_LibraryCard()` | | | | × | × | |
| `reserve_Book()` | | × | × | × | | |
| `sort_Catalog()` | × | × | | × | | × |

**Table 1.** Scaling of the Blob context along the relation `call`. For space limitation, individuals that are not affected by relational scaling are omitted.

For example, assume methods are scaled along relation *call* regarding the lattice of the context in the left hand side of Fig. 2, which is composed of the concepts $\{c0, c2, c4, c5, c6, c11\}$ and the respective precedence links illustrated in Fig. 3. Since the method `sort_Catalog()` calls the method `add_Book()` which appears in the extent of concepts $c0$, $c2$ and $c5$ and calls the method `remove_Book()` which belong to the extent of concepts $c11$ and $c5$, the Blob context is extended by the relational properties *call:c0*, *call:c2*, *call:c5* and *call:c11*. Table 1 presents the integration of the relation *call* to the Blob context.

The scaling mechanism is only one step in the global analysis process which, given a RCF, yields a set of lattices, one per context, called *relational lattice family* (RLF). The RLF is defined as the set of lattices whose concepts jointly reflect *all* the shared properties and links among individuals of the RCF. Its construction is an iterative process because the scaling mechanism modifies contexts and thereby the corresponding lattices, which in turn may require a new scaling to reflect the newly formed concepts and the link sharing they provoke. Iterations stop whenever a fixed point is reached, *i.e.*, further scaling leaves all the lattices in the RLF unchanged.

Lattice evolution is illustrated though the analysis of the Blob RCF in Fig. 2: RCA yields the concept lattice illustrated in Fig. 3. The final lattice of the Blob is different from the initial one due to the relational information inserted into the scaled version of the Blob context. Indeed, the individuals are assigned relational properties that lead to the sharing of more properties among these individuals. By factoring out the new properties into concept intents, links between individuals are lifted up to the concept level, yielding relations between concepts[6]. Thus, in Fig. 3, previously existing concepts obtain new properties while completely new concepts emerge. For example, the concept $c16$ that represents the method `sort_catalog()` has been assigned the relational properties `call:c0` and `call:c11`, which means that `sort_catalog()` calls methods in the extent of concept $c0$ and $c11$, namely `add_book()` and `remove_book()`. Furthermore, methods `borrow_Book()` (concept $c3$) and `reserve_Book()` (concept $c12$) have top concept as immediate successor in the initial lattice. Their link with the method `check_Availability_Book()` (concept $c4$) has been revealed through scaling. They form a new concept

[6] Observe that for compactness reasons, only non-redundant relational properties are visualized in concept intents, *i.e.*, those referring to the most specific concepts.

$c19$ (see Fig. 3) that represents the set of methods that call `check_Availability_Book()`.

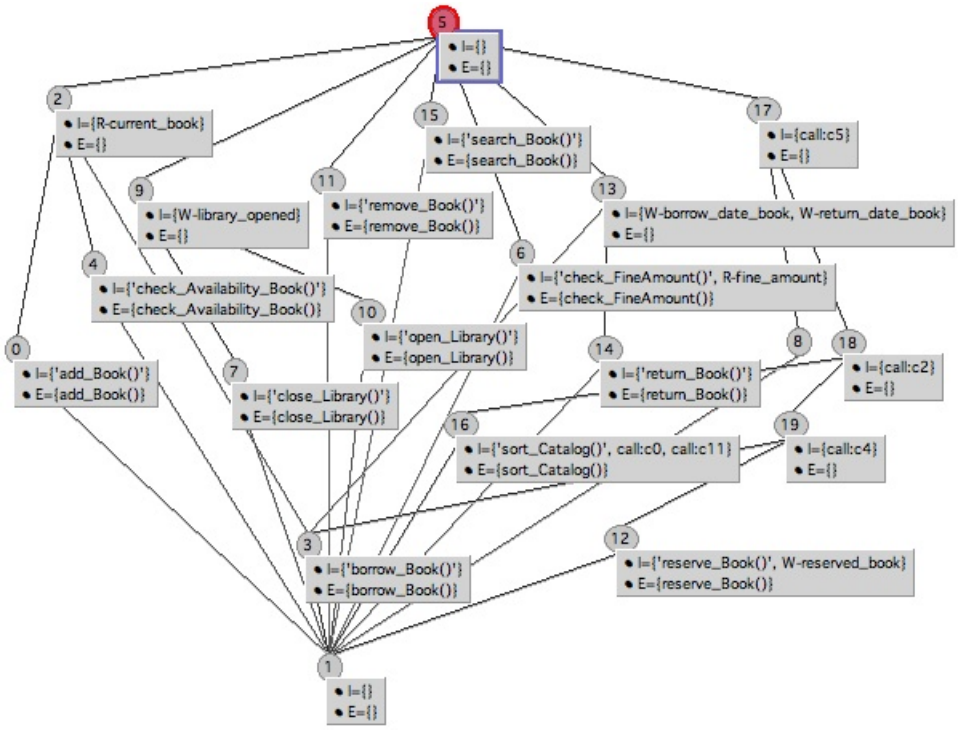


**Fig. 3.** The lattice of the context of methods shown in Fig. 2.

## 4 Correction of Design Defects using RCA

Our intuition is that design defects resulting in high coupling and low cohesion could be improved by redistributing class members among existing or new classes to increase cohesion and/or decrease coupling. RCA provides a particularly suitable framework for the redistribution because it can discover strongly related sets of individuals with respect to shared properties and inter-individual links and hence supports the search of cohesive subsets of class members. Fig. 4 depicts our approach for the identification of refactorings to correct design defects in general and the Blob in particular. It shows the tasks of detection of design defects and of correction of user-validated defects.

### 4.1 Overall process

We define a three-step RCA-based correction process that follows a two-step defect detection process. First, we build a model of the program that is simpler to manipulate than the raw source code and therefore eases the subsequent activities of detection and correction. The model is instantiated from a metamodel to describe OO programs. Next, we apply well-known algorithms based on metrics and–or structural data on this model to single out suspicious classes having potential design defects [17]. For each suspicious class, we automatically extract a RCF that encodes relationships among class members from the model of the program. Then, the obtained RCF is fed into a RCA engine that derives the corresponding concept lattices. Finally, the discovered concepts are explored using some simple algorithms, which apply a set of refactoring rules that allow the identification of cohesive sets of fields and methods. The approach suggests a set of refactorings that jointly amount to splitting the Blob into as many classes as there are cohesive sets and merge the content of the surrounding classes with the new classes whenever appropriate.

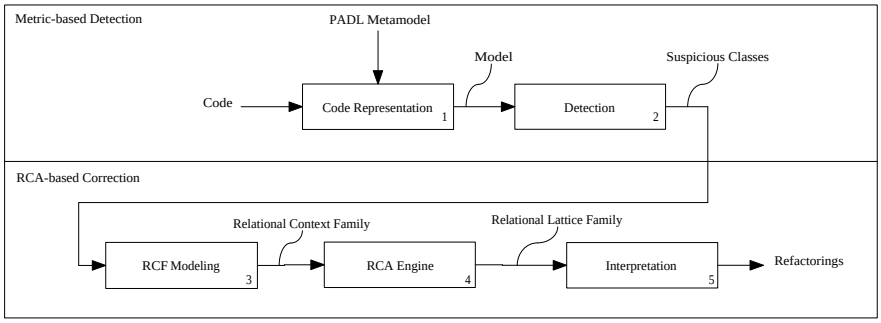


**Fig. 4.** RCA-based Workflow for the Detection and Correction of Design Defects.

### 4.2 RCF Extraction

To correct design defects, we need to identify cohesive sets of methods with respect to the mode of usage of fields, *i.e.*, read or write, and call between methods. Hence, the individuals are methods of the large class and properties are its fields. The incidence relation represent the access of fields in read/write mode. In order to differentiate between the two access modes, the prefixes `R-` and `W-` are added to the name of the fields as illustrated in Fig. 2. Method invocations within the large class are encoded by a dedicated inter-individuals relation denoted *call* (see the table in the right hand side of Fig. 2).

The formal attributes were derived from names of methods and added to the method context. These attributes allow the emergence of a single concept for each

method, called *method concept*[7], in the corresponding lattice. Beside listing the entire set of properties of a given method, the concept method helps preserving one-to-one invocation between methods. These details can be lost during the scaling step that aims at integrating the relation *call* into the context of the large class by substituting one-to-many invocations for those of type one-to-one.

### 4.3 Deriving the lattice

Fig. 3 represents the concept lattice obtained by the RCF engine from the context given in Fig. 2. The concepts of the lattice represent the refactoring opportunities of the design defect. Indeed, concepts such as $c9$ exhibit group of methods using the same sets of fields and fields used by cohesive sets of methods. These concepts are considered as class candidates because they are cohesive. In addition, concepts such as $c3$ and $c12$ highlight subsets of cohesive methods, because methods calling the same set of other methods are highly cohesive. A third category of concepts such as $c9$ and $c13$ represent the *use-relationship* between methods of the large class and the surrounding data classes. The study of these concepts allow to assess the coupling between the large class and its surrounding data classes. Thus, we can identify which methods and fields of the large class should be moved to surrounding classes.

### 4.4 Suggesting Refactorings

The RLF of the Blob is used to interpret the inner structure of the Blob and then suggest refactorings. More specifically, we apply algorithms looking for concepts that reflect the presence of highly cohesive and weakly coupled sets. Intuitively, shared usages of fields and calls of methods is a sign of cohesion whereas coupling is directly expressed by the reliance of a method on a surrounding class (method and-or field). Following these design guidelines, we correct the Blob in two ways. First, we move disjoint and cohesive sets of methods and-or fields that are related to a data class in that data class. Two refactorings describe such migration between classes: *Move Method* [5, p.142] and *Move Field* [5, p.146]. Second, we organise cohesive subsets that are not related to data classes in separate classes. In addition to the two previous refactorings, we use the refactoring *Extract Class* [5, p.149], which consists in creating a new class and moving the chosen fields and methods from the old class to the new class using the two first previous refactorings.

We have specified three refactoring rules to build incrementally cohesive sets by visiting the concept lattice of methods. These rules are applied in sequence, *i.e.*, we apply the two first rules that deal with the access of fields by methods in read-write mode and then the rule that handle method calls.

***Rule 1.*** Methods accessing in write mode the same set of fields are gathered in a single cohesive set.

[7] The smallest extent in the lattice containing this method.

***Rule 2.*** Methods accessing in read mode the same set of fields are gathered in a single cohesive set if the number of common fields that they access is higher than the number of fields they access separately.

These two rules are inspired from the object identification approach described in [23] where grouping of methods is based on the accessed fields, with respect to the number of fields they access separately. The obtained cohesive sets are merged according to the following rule:

***Rule 3.*** Methods that call the same set of methods are put in a single cohesive set if the number of jointly called methods is higher than the number of methods called separately.

For example, by applying the three previous rules on the running example of the library Blob class, we obtain several cohesive sets as illustrated in Fig. 5, on the left. The cohesive sets that should be migrated in the data classes are shown in Fig. 5 on the right. This last step is currently performed manually but planned to be automated.

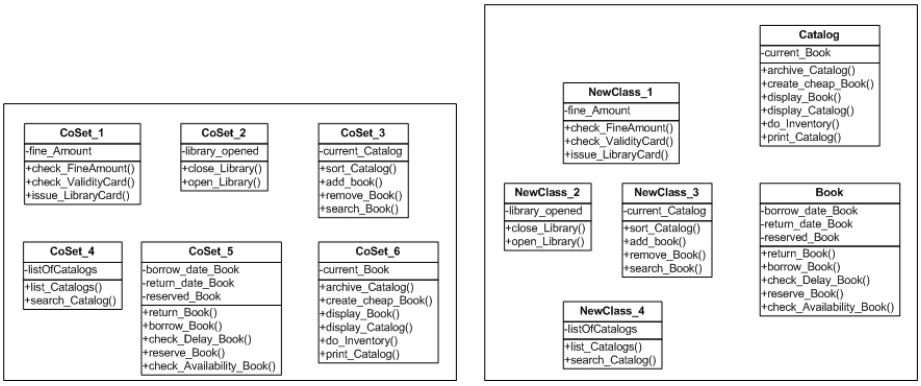


**Fig. 5. Left:** The cohesive sets obtained from class `Library_Main_Control` depicted in Fig. 1 . **Right:** Moving these cohesive Sets to existing data-classes or new-classes.

We provide the implementation details of these rules in the following.

***Implementing Rule 1.*** We iterate the lattice and record all concepts related to fields with the prefix `W-`. We mark all these concepts as visited. We sort this list in reverse order by the number of fields with `W-`. Thus, fields that are accessed in write mode by a high number of methods are processed first. For example, the concept $c3$ in Fig. 3 is processed first because of the related concept $c13$. For each concept of the list, we create a new cohesive set and apply the method applyRuleWrite(). This method consists in moving the current(s) field(s) (`borrow_date_book` et `return_date_book` in concept $c13$) with the refactoring

*Move Field* and for each method in the intent of the current concept (`borrow_Book()`) that has not yet been included in a set (*i.e.*, not yet visited), we move it to the current cohesive set using the refactoring *Move Method*. Then, recursively, we check the parents of the current concept and the children of a parent if interesting to explore. The children of a parent are interesting to explore if the parent contains at least one `W-` field also contained in the current concept. For example, only the children of the parent $c13$ of the concept $c3$ are interesting to explore. We reapply the rule applyRuleWrite() on the children.

***Implementing Rule 2.*** This rule consists in finding the best cohesive set of methods that access to a common set of fields in read mode. For each concept related to common fields in read mode and not yet visited *i.e.*, not processed when applying Rule 1, and thus not included in a set, we calculate a ratio. The ratio corresponds to the number of fields in common with their total number of fields. We calculate the mean of all the ratios corresponding to each concept and retain only groups of concepts that have a mean higher than 0.5, *i.e.*, concepts whose methods accessing a common number of fields is higher than their own number of fields in average. We obtain thus a list of candidate sets of concepts that we sort in reverse order to process first concepts with a greater ratio. For each sets of concepts, we create a new cohesive set by moving the methods and fields with the respective appropriate refactorings (*Move Field* and *Move Method*).

***Implementing Rule 3.*** This rule is similar to rule 2. The difference is that we identify common methods called by one or several methods of the resulting cohesive sets built from Rules 1 and 2. We calculate also a ratio and select the best candidates, and then merge the cohesive sets according to the value of their ratio.

## 5 Experimental Study

We use PADL [10] to model source code and Galicia v.2.1 [20], to construct and visualize the concept lattices. PADL is the meta-model at the heart of the Ptidej tool suite (*Pattern Trace Identification, Detection, and Enhancement in Java*) [8]. Galicia is a multi-tool open-source platform for creating, visualizing, and storing concept lattices [20]. Both tools communicate by means of XML files describing data and results. Thus, an add-on to Ptidej generates contexts in the XML format of Galicia, which are then transformed by the tool into lattices and shown on screen for exploration.

In order to validate the proposed approach for the detection and correction of Blob design defects, we consider four different open-source programs. We use *freely available* programs to ease comparisons and replications of our experiments. We provide some information on these programs in Table 2

In Azureus, we found 41 Blobs by applying our detection algorithms. We notice that the underlying classes are difficult to understand, maintain, and

| Name | Version | Lines of Code | Number of Classes | Number of Interfaces |
|---|---|---|---|---|
| AZUREUS | 2.3.0.6 | 191,963 | 1,449 | 546 |
| A peer-to-peer client implementing the BitTorrent protocol | | | | |
| LOG4J | 1.2.1 | 10,224 | 189 | 14 |
| A logging Java package | | | | |
| LUCENE | 1.4 | 10,614 | 154 | 14 |
| A full-featured text-search Java engine | | | | |
| NUTCH | 0.7.1 | 19,123 | 207 | 40 |
| An open-source web search engine, based on LUCENE | | | | |

**Table 2.** List of Programs.

reuse because they have a large number of fields and methods. For example, the class `DHTTransportUDPImpl` in the package `com.aelitis.azureus.core.dht.transport.udp.impl`, which implements a distributed sloppy hash table (DHT) for storing peer contact information over UDP, has an atypically large size. It declares 42 fields and 66 methods for 2,049 lines of code. It has a medium-to-high cohesion of 0.542 and a high coupling of 81 ($8^{th}$ highest value among 1,626 classes). The data classes that surround this large class are: `Average`, `HashWrapper` in package `org.gudy.azureus2.core3.util` and `IpFilterManagerFactory` in package `org.gudy.azureus2.core3.ipfilter`.

Table 3 provides the results of applying our rules on three different Blobs classes detected in Azureus and on two Blobs classes in the three other programs. It is noteworthy that the results provided by our method have been assessed manually: Among the set of all cohesive sets in the output we identified those whose semantics could be clearly established and it confirmed their cohesiveness. A measure for the precision of our method is the ratio of the real cohesive sets to the total number of sets output by the method. As Table 3 indicates, the precision may vary within a wide range (from 30 to 70 % of correct guesses). The cohesive sets suggested by our approach include an important number of small cohesive sets, which include generally at most one field and one or two methods. This explains why we did not get a good precision. The other concise sets gather between 10 and 20 fields/methods and are good candidates for the creation of new classes because they define a specific responsibility or semantics.

To increase the robustness of our approach, we need to define additional rules related to the access of fields and methods by methods not only within one class but also located in other associated classes. Moreover, our analysis is purely static. Thus, we need to enhance our method with a dynamic analysis to preserve the behavior of the program. Finally, the restructuring should be semi-supervised by an expert because only experts could assess the relevance of grouping elements. The method should be seen as a support for restructuring huge number of data. Thus, we share Snelting's opinion that an interactive restructuring performed by the software engineer is more appropriate [24].

| System | Blob Class | Size (number of fields and methods) | Lines of code | Cohesion | Coupling | Number of fields and methods moved | Number of cohesive sets | Number of real cohesive sets |
|---|---|---|---|---|---|---|---|---|
| Azureus v2.3.0.6 | `DHTTransportUDPImpl` | (42+66) 108 | 2,049 | 0.542 | 81 | (27+32) 59 | 10 | 7 |
| | `DHTControlImpl` | (47+80) 127 | 1,868 | 0.52 | 67 | (35+62) 97 | 19 | 11 |
| | `TRTrackerBTAnnouncerImpl` | (36+47) 83 | 1,393 | 0.948 | 54 | (24+33) 57 | 16 | 5 |
| Log4j v1.2.1 | `LogBrokerMonitor` | (29+105) 134 | 1,591 | 0.479 | 86 | (23+85) 108 | 31 | 17 |
| | `Category` | (9+53) 62 | 1,042 | 0.831 | 46 | (8+44) 52 | 18 | 9 |
| Lucene v1.4 | `IndexReader` | (7+52) 59 | 593 | 0.661 | 68 | (5+30) 35 | 4 | 2 |
| | `QueryParser` | (36+48) 84 | 1,085 | 0.3 | 26 | (24+37) 61 | 13 | 10 |
| Nutch v0.7.1 | `FSNamesystem` | (24+35) 59 | 1,211 | 0.908 | 23 | (17+25) 42 | 18 | 9 |
| | `JobTracker` | (22+31) 53 | 910 | 0.938 | 21 | (17+18) 35 | 11 | 8 |

**Table 3.** Blob Classes in Four Different Programs and the Number of Cohesive Sets

## 6 Related Work

Few studies have explored the semi-automatic correction of design defects. Thus, we only list work related to design defects and to the use of FCA in software maintenance.

Sahraoui et *al.* in [23] proposed an approach for identifying objects in *procedural* code, a problem that is similar to the split of a Blob (in this case the Blob corresponds to the entire application or to a module thereof). The approach combines metrics calculation with several FCA-based analysis steps in class identification and further graph-based reasoning to detect associations among newly identified classes.

Snelting and Tip [24] proposed a FCA-based method for adapting a class hierarchy to a specific usage thereof. It comprises a study of the way class members are used in the client code of a set of applications. The study enables the identification of anomalies in the design of class hierarchies, *e.g.*, class members that are redundant or that can be moved into a derived class. In contrast, we detect design defects at a higher level as specified in the literature. Moreover, beyond pure hierarchies, we are interested in classes with associations.

Godin and Mili [7] used concept lattices for class hierarchy redesign based on classes signatures. Yet like [11], they find useful hierarchy restructuring and member redistribution but ignore any possible relationships among the members of a class.

Marinescu [16] presented an approach based on *detection strategies* which applies metrics computation. Combinations of metrics through filtering and composition are used to capture deviations from good design principles and heuristics. Yet the method is inherently limited as design flaws admit no easy detection exclusively by metrics: the *structure* of a design matters and it is impossible to capture in numbers. In contrast, our approach relies on a combination of metrics

for the detection of design defects with a clustering and visualisation technique, FCA, that allows the design structure to be fully comprehended.

The work of Kirk *et al.* [12] comes close to ours. Yet they use attribute slicing to refactor large classes, *i.e.*, they slice the variable set of the class into subsets based on the usage of variables by methods. The approach was designed to deal with the Large Class code smell and hence has a scope of a single class whereas Blob involves multiple classes. Conversely, they use an intra-method slicing techniques that allows the precise set of instruction manipulating a instance variable to be detected and the isolated. The practical validation of the approach is yet to be done.

Tonella and Antoniol used FCA to infer recurring patterns in program models [26]. Their study yielded impressive results in terms of groups of classes having common structural relations. However, their approach seems of limited interest for us because it detects only structural relations, whereas design defects are often characterised by measurable properties (*e.g.*, a large class has *a large number* of fields and methods). FCA is not devised to deal with numerical measurement, hence it could benefit from metrics-based techniques.

Arévalo *et al.* [1] applied FCA to identify implicit dependencies among classes in program models (extracted from source code). A set of views at different levels of abstraction are built: At the class level, views show the access of methods to variables and the patterns of calls among methods in a class, hence they help to assess the class cohesion. At the class hierarchy level, views highlight common and irregular forms of hierarchies so as to deduce possible refactorings. At the program level, they refined and extended the approach of Tonella *et al.* to any (recurring) regularities such as design patterns, architectural constraints, idioms, etc. Our approach is similar in that it detects flaws, but our choices of the elements and properties to be analysed are guided by the descriptions of the defects.

## 7 Conclusion

We proposed an approach that uses RCA to suggest appropriate refactorings to correct certain design defects. In particular, we showed how our approach can help refactoring programs with Blob design defects. Unlike other FCA-based restructuring approaches, we worked on whole lattice regions rather than on separate concepts because candidate refactoring are inferred from several concepts in the lattice. We illustrated our approach using an example of a Library management system and validated it on Azureus v2.3.0.6 and three other programs. We showed that using RCA, our approach could suggest relevant refactorings to improve the program. The generalisation of our results to other design defects is briefly discussed and will be developed in future work. Future work will also include assessing more programs via our approach and discussing the proposed refactorings with their developers and apply them. We also plan to performed quantitative studies on the trade-off between cohesion and coupling.

## References


1. G. Arévalo. *High Level Views in Object Oriented Systems using Formal Concept Analysis*. PhD thesis, University of Berne, Janv 2005.
2. J. V. Bart Du Bois and S. Demeyer. Refactoring - improving coupling and cohesion of existing code. In *Proceedings of WCRE*, pp 144–151, 2004.
3. S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.
4. N. Fenton and S. L. Pfleeger. *Software metrics (2nd ed.): a rigorous and practical approach*. PWS Publishing Co., 1997.
5. M. Fowler. *Refactoring – Improving the Design of Existing Code*. Addison-Wesley, $1^{st}$ edition, 1999.
6. R. Godin and P. Valtchev. Formal concept analysis-based normal forms for class hierarchy design in oo software development. In *Formal Concept Analysis: Foundations and Applications*, chapter 16, pp 304–323. Springer Verlag, 2005.
7. R. Godin and H. Mili. Building and maintaining analysis-level class hierarchies using galois lattices. In *Proceedings of OOPSLA*, pp 394–410, 1993.
8. Y.-G. Guéhéneuc. A reverse engineering tool for precise class diagrams. In *Proceedings of CASCON*, pp 28–41, 2004.
9. Y.-G. Guéhéneuc and H. Albin-Amiot. Using design patterns and constraints to automate the detection and correction of inter-class design defects. In *Proceedings of TOOLS*, pp 296–305, 2001.
10. Y.-G. Guéhéneuc and H. Albin-Amiot. Recovering binary class relationships: Putting icing on the UML cake. In *Proceedings of OOPSLA*, pp 301–314, 2004.
11. Marianne Huchard and Hervé Leblanc. Computing interfaces in java. In *Proceedings of ASE*, pp 317–320, 2000.
12. D. Kirk, M. Roper, and N. Walkinshaw. Using attribute slicing to refactor large classes. In *Seminar Proceedings of Beyond Program Slicing*, number 05451, 2006.
13. M. Lanza. CodeCrawler—Lessons learned in building a software visualization tool. In *Proceedings of CSMR*, pp 409–418, 2003.
14. M. Lanza. *Object-Oriented Reverse Engineering – Coarse-grained, Fine-grained, and Evolutionary Software Visualization*. PhD thesis, Institute of Computer Science and Applied Mathematics, May 2003.
15. R. Marinescu. *Measurement and Quality in Object-Oriented Design*. PhD thesis, Politehnica University of Timisoara, Oct 2002.
16. R. Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *Proceedings of ICSM*, pp 350–359, 2004.
17. N. Moha, Y.-G. Guéhéneuc, and P. Leduc. Automatic generation of detection algorithms for design defects. In *Proceedings of ASE*, 2006.
18. N. Moha, J. Rezgui, Y.-G. Guéhéneuc, P. Valtchev, and G. El Boussaidi. Using FCA to suggest refactorings to correct design defects. In *Proceedings of CLA*, 2006.
19. Object Technology International / IBM. Eclipse platform – A universal tool platform, 2001.
20. Galicia, Sept 2005. http://sourceforge.net/projects/galicia/.
21. D. Raţiu, S. Ducasse, T. Gîrba, and R. Marinescu. Using history information to improve design flaws detection. In *Proceedings of CSMR*, pp 223–232, 2004.
22. A. J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.
23. H. A. Sahraoui, H. Lounis, W. Melo, and H. Mili. A concept formation based approach to object identification in procedural code. In *Automated Software Eng.*, 6(4):387–410, 1999.

24. G. Snelting and F. Tip. Understanding class hierarchies using concept analysis. *ACM TOPLAS*, 22(3):540–582, 2000.
25. Azureus, June 2003. http://azureus.sourceforge.net/
26. P. Tonella and G. Antoniol. Object oriented design pattern inference. In *Proceedings of ICSM*, pp 230–240, 1999.
27. A. Trifu and I. Dragos. Strategy-based elimination of design flaws in object-oriented systems. In *Proceedings of WOOR*, 2003.
28. W. Brown, R. Malveau, H. McCormick, and T. Mowbray. *AntiPatterns Refactoring Software, Architectures, and Projects in Crisis*. Robert Ipsen, 1998.